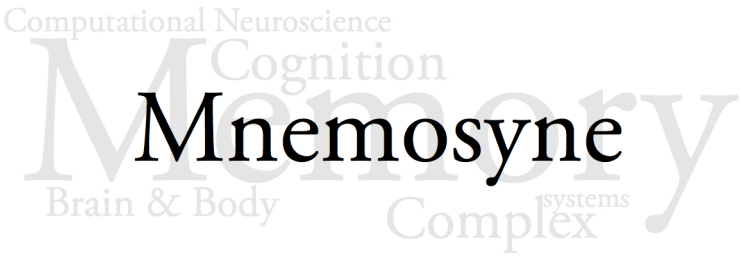
université de BORDEAUX



Rapport de stage licence 3 mention Informatique
Université de Bordeaux

Ajout d'apprentissage bio-inspiré dans un *Echo State Network*

Maître de stage : Xavier Hinaut
Équipe : Mnemosyne
Laboratoire : Inria Bordeaux - Sud-Ouest

Rémy Portelas

26 juin 2016

Table des matières

1 Remerciements

Je tiens à remercier l'*Inria* pour m'avoir permis d'effectuer un stage dans leurs locaux. Je remercie également l'équipe *Mnemosyne* pour m'avoir accueilli et tout particulièrement mon maître de stage, Xavier Hinaut, pour son encadrement et ses explications.

2 Introduction

Mon sujet de stage était d'ajouter des méthodes d'apprentissage non-supervisé dans un Réseau de Neurones Artificiels (RNA). Les modifications étaient à effectuer sur un *Echo State Network* (ESN), un type de RNA ne possédant en temps normal que des méthodes d'apprentissage supervisés. Pour me préparer au mieux, j'ai suivi au préalable une formation en ligne dispensée gratuitement par l'Université de Stanford sur l'Apprentissage Automatique. Durant ce stage j'ai pris en main un code minimaliste d'ESN que j'ai étoffé en lui ajoutant une méthode d'apprentissage basée sur la plasticité intrinsèque des cerveaux biologiques. Par la suite je me suis inspiré de ce code minimaliste pour implémenter un nouvel ESN ayant pour objectif d'apprendre implicitement les règles de la grammaire d'Elman. Enfin j'ai étudié l'impact de l'apprentissage par plasticité intrinsèque dans ce nouvel ESN.

3 Présentation de l'équipe-projet *Mnemosyne*

Les travaux de recherche de l'équipe *Mnemosyne* combinent les neurosciences, l'informatique (plus particulièrement l'apprentissage automatique) et la robotique. Leur objectif est de modéliser le cerveau comme un ensemble de mémoires actives intéragissant entre elles et avec le monde extérieur. Le système ainsi créé est incarné et situé dans un robot réel ou virtuel pour tester ses comportements autonomes.

4 Généralités sur les réseaux de neurones artificiels

Les réseaux de neurones artificiels ont été inventés dans les années 50 par McCulloch et Pitts. Ses outils mathématiques, inspirés par les réseaux de neurones biologiques, permettent d'approximer des fonctions ou des systèmes dynamiques temporels qui peuvent dépendrent de nombreux paramètres. Les réseaux de neurones peuvent être divisés en deux catégories principales :

les RNAs *supervisés* et *non-supervisés*. La description qui suit définit les différentes variantes des RNAs *supervisés*.

Un neurone artificiel peut être vu comme une fonction qui prend un vecteur d'entrées u pondérées par un vecteur de poids et retourne une activation y. C'est l'interconnection de ce genre de neurones qui est appelé réseau de neurones.

Avant d'être utilisé un RNA doit passer par une phase d'apprentissage. Un ensemble de couples *(vecteur d'entrée, sortie attendue)* constitue les exemples d'entraînement. Chaque vecteur d'entrée est donné au réseau qui propage l'information jusqu'à la couche de neurones de sortie dont l'activation constitue une prédiction de résultat. C'est l'étude de l'erreur entre la prédiction et la sortie attendue qui permet de modifier le réseau pour qu'il réponde plus précisément. L'idée est de modifier les vecteurs de poids du réseau pour que sa sortie se rapproche le plus possible de la sortie désirée.

Deux types de problèmes peuvent être résolus par les RNAs :

— Les problèmes de régression où l'objectif est d'approximer une fonction comme sur la figure 4 (approche continue : Les activations du réseau sont des nombres réels directement interprétés comme les prédictions du réseau).

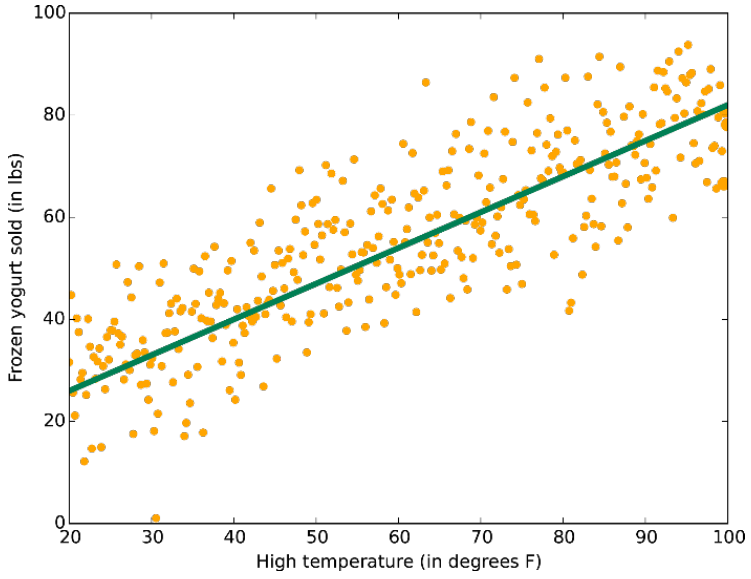


FIGURE 1 – Approximation de la vente de yahourts frais selon la température. La droite verte représente une hypothèse linéaire qu'un RNA pourrait estimer à l'aide des couples d'entraînement (température, yahourt vendus) représentés par les points jaunes.

— Les problèmes de catégorisation où il est question d'associer à chaque entrée la classe qui lui correspond, comme sur la figure 2. On peut par exemple penser à un RNA qui, si on lui donne un mail en entrée, donne en sortie une prédiction de sa catégorie qui pourrait être Spam ou Non-Spam. La catégorisation est une approche discrète du problème car les sorties possibles du réseau sont finies, elles correspondent au nombre de catégories du problème.

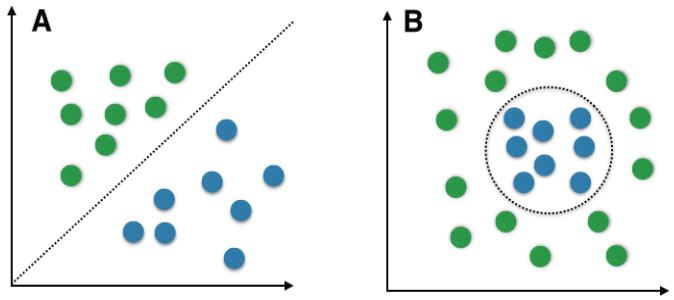


FIGURE 2 – A : la ligne en pointillé représente une hypothétique catégorisation linéaire des ronds vert et bleu. B : le cercle en pointillé représente une catégorisation non linéaire des ronds verts et bleus.

Bien que les types de RNAs *supervisés* soient très variés, on peut les séparer en deux classes majeures : les RNAs *feed-forward* et les RNAs *récurrents*.

4.1 Réseaux *feed-forward*

Les RNAs *feed-forward* (voir figure 4.1) sont arrangés en plusieurs couches de neurones. Le modèle de base est en général composé de 3 types de couches : une couche d'entrée, une ou plusieurs couches cachées puis une couche de sortie. Le réseau est dit *feed-forward* car un neurone de couche n ne peut être connecté qu'à un neurone de la couche $n+1$.

Ces types de RNAs sont majoritairement utilisés de nos jours. De nombreuses publications en apprentissage automatique leurs sont dédiées. Il existe beaucoup d'applications aux RNAs *feed-forward* comme :

— Reconnaissance d'écriture manuscrite
— Reconnaissance d'images
— Conduite automatisée de véhicules
— Catégorisation de mails
— ...

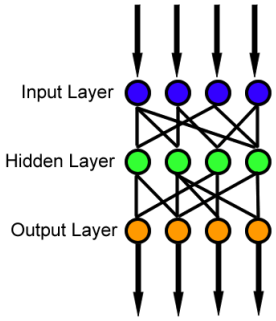


FIGURE 3 – Réseau de neurones *feed-forward*. L'activation des neurones est effectuée couche par couche : Le signal d'entrée est donné à la couche d'entrée, son activation est alors transmise à la couche cachée(ici il n'y en a qu'une seule). L'activation de la couche cachée constitue l'entrée de la couche de sortie dont l'activation représente l'hypothèse du réseau pour le signal d'entrée donné.

4.2 Réseaux *récurrents*

Les RNAs *récurrents* (voir figure 4) sont constitués d'un ou plusieurs cycles au niveau des connexions entre leurs neurones. Cette nature récurrente leur permet de résoudre des tâches avec une notion temporelle comme par exemple la prédictions (climatique, financière, ...), la gestion des mouvements en robotique ou encore la reconnaissance vocale.

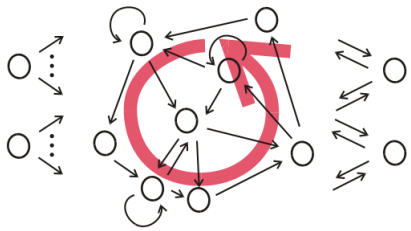
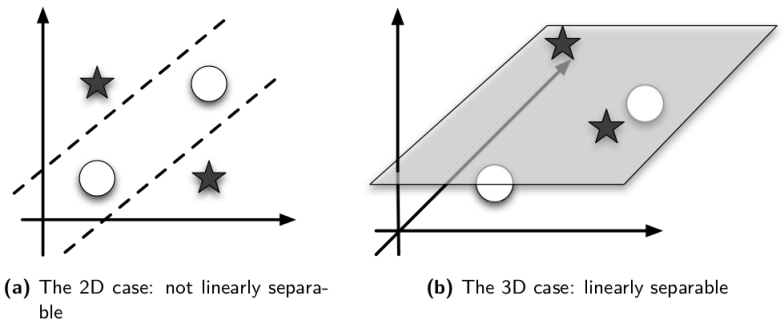


FIGURE 4 – Réseau de neurones *récurrent*. Le signal d'entrée est donné à la couche d'entrée (noeuds les plus à gauche). L'activation de cette couche est alors transmise à la couche cachée, représentée par les noeuds interconnectés au centre du graphique. Cette couche possède un ou plusieurs cycles, visibles au niveau des flêches entre les noeuds. L'activation de la couche cachée est donnée à la couche de sortie qui produit l'hypothèse de prédiction du réseau.

4.3 *Echo State Network*

Le type de réseau étudié dans ce stage, l'ESN, est un réseau *récurrent*. La partie cyclique d'un ESN (la couche cachée) est souvent nommée le réservoir, c'est un ensemble de neurones interconnectés dont les poids, après leur initialisation ne seront pas modifiés durant la phase d'apprentissage surpervisé. Ce réseau présente un intérêt en terme d'efficacité, en effet concernant sa phase d'apprentissage supervisé, seule la matrice de sortie est entrainée par régression linéaire (complexité en $O(n)$ par rapport à la taille du réservoir pour les méthodes d'apprentissage supervisé en ligne, contrairement aux réseaux de neurones *feed-forward* classiques qui utilisent la méthode nommée *Backpropagation* dont la complexité est $O(n^2)$ d'après les travaux de Jaeger [1]).

La force de ce réseau pouvant résoudre des problèmes non linéaires avec pour seul apprentissage une régression linéaire réside dans son réservoir. Lorsqu'on active les neurones d'entrée du réservoir avec le vecteur d'entrée du RNA, on effectue finalement une projection dans un espace non-linéaire de haute dimension. Si on prend l'exemple simpliste du problème de catégorisation de la fonction XOR énoncé dans l'article de D. Verstraeten [2] on peut voir que dans la figure 5 (a) il est impossible de trouver une unique droite séparant les ronds des étoiles. Cependant une fois projeté en 3D dans la figure 5 (b) il est possible de résoudre linéairement le problème (grâce à un hyperplan séparateur).



(a) The 2D case: not linearly separable

(b) The 3D case: linearly separable

FIGURE 5 – Projection de l'entrée dans un espace non-linéaire de plus haute dimension pour pouvoir linéairement résoudre le problème de catégorisation

L'ESN utilise parfois des neurones dits *à fuites* : le résultat de la fonction d'activation d'un neurone à l'instant t va alors dépendre de son vecteur d'entrée à l'instant t mais aussi de son activation à l'état *t-1*. Ils agissent

comme des condensateurs, leurs fluctuations sont progressives. Ils permettent de pourvoir le réservoir d'une mémoire à court terme. La description formelle du type d'ESN étudié au cours de ce stage se fera dans la section suivante.

5 Méthodes et problèmes considérés

Après m'être familiarisé avec les concepts théoriques des RNAs, un code minimaliste d'ESN approximant une série chaotique en Python, réalisé par Mantas Lukoševičius et amélioré par Xavier Hinaut, m'a été fourni. J'avais pour objectif d'ajouter de la Plasticité Intrinsèque (PI) dans le réseau pour en tester l'impact sur les performances au niveau de la prédiction de la série chaotique de Mackey-Glass. J'ai aussi créé un autre ESN apprenant implicitement les règles de la grammaire d'Elman. Enfin j'ai ajouté de la PI dans cet ESN pour étudier les conséquences sur les performances d'apprentissage de la grammaire.

Après une formalisation de l'ESN utilisé, je décrirai comment celui-ci a été appliqué aux tâches étudiées. La formalisation qui va être donnée est une traduction simplifiée des travaux de Mantas Lukoševičius [3].

5.1 Formalisation

On peut utiliser un ESN lorsqu'on dispose de couples composés d'un signal d'entrée $u(n) \in \mathbb{R}^{N_u}$ et de la sortie désirée $y^{target}(n) \in \mathbb{R}^{N_y}$ avec $n = 1, 2, ...T$, T étant le nombre de couples (x,y) du jeu d'entrainement. L'objectif est d'entrainer le réseau pour qu'il produise des prédictions $y(n) \in \mathbb{R}^{N_y}$ les plus proches possible de $y^{target}(n)$. Pour mesurer l'erreur $E(y, y^{target})$ de la prédiction par rapport à la sortie désirée on utilise ce qui est communément appelé la méthode des moindres carrés.

$$E(y, y^{target}) = \frac{1}{N_y} \sum_{i=1}^{N_y} \sqrt{\frac{1}{T} \sum_{i=1}^{T} (y_i(n) - y_i^{target}(n))^2} \tag{1}$$

Les équations de mise à jour du réseau sont

$$x^{'}(n) = \tanh(W^{in}[1; u(n)] + Wx(n-1)), \tag{2}$$

$$x(n) = (1-\alpha)x(n-1) + \alpha x^{'}(n), \tag{3}$$

où $x(n) \in \mathbb{R}^{N_x}$ est le vecteur des activations des neurones du réservoir et $x^{'}(n) \in \mathbb{R}^{N_x}$ est sa mise à jour à l'instant n. La fonction tangente hyperbolique tanh(.) est appliqué pour chaque élément et [. ; .] représente une

concaténation verticale de vecteurs (ou matrice), ici cela matérialise l'ajout d'une constante. $W^{in} \in \mathbb{R}^{N_x \times (1+N_u)}$ et $W \in \mathbb{R}^{N_x \times N_x}$ sont respectivement les poids des entrées et des connexions récurrentes dans le réservoir, $\alpha \in [0, 1]$ représente le ratio de fuite des neurones.

La sortie du réseau est obtenue de la manière suivante :

$$y(n) = W^{out}[1; u(n); x(n)], \tag{4}$$

où $y(n) \in \mathbb{R}^{N_y}$ est la sortie du réseau, $W^{out} \in \mathbb{R}^{N_y \times (1+N_u+N_x)}$ est la matrice des poids de sortie, et $[.\,;.\,;.]$ représente une fois de plus une concaténation verticale de vecteurs (ou matrices).

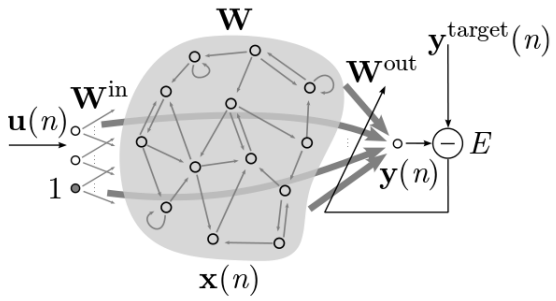


FIGURE 6 – Un *Echo State Network*

L'utilisation de ce genre d'ESN (voir figure 6) peut se résumer aux quatre étapes suivantes :

1. générer un réservoir aléatoire (W^{in}, W). Cette partie est l'étape la plus délicate dans l'utilisation d'un ESN car le réservoir dépend de nombreux hyper-paramètres (taille du réservoir, rayon spectral, ...), leurs rôles sont explicités en détails dans l'article de Mantas Lukoševičius [3] ;
2. activer le réservoir à l'aide du jeu d'entrainement $u(n)$ et collecter les activations du réservoir $x(n)$ correspondantes ;
3. calculer les poids de sortie linéaire W^{out} à l'aide d'une régression linéaire en minimisant l'erreur (calculé avec la méthode des moindres carrés) entre $y(n)$ et $y^{target}(n)$;
4. utiliser le réseau avec les poids entrainés W^{out} sur de nouvelles entrées $u(n)$ pour calculer l'approximation $y(n)$ correspondante.

Une expérience a été réalisée sur le rayon spectral du réservoir, il convient donc de donner quelques éclaircissements quant à son rôle. Le rayon spectral

est un hyper-paramètre du réservoir permettant d'influer sur l'amplitude de la distribution des éléments non-nuls de la matrice W (les connections récurrentes du réservoir). Le rayon spectral $\rho(W)$ est la plus grande valeur propre d'une matrice W. Après avoir calculé le rayon spectral de W, on divise W par $\rho(W)$ pour obtenir une matrice avec un rayon spectral unitaire. On multiplie enfin à notre convenance la matrice obtenue avec l'hyper-paramètre du réservoir nommé rayon spectral. Un grand rayon spectral augmente l'excitabilité des dynamiques du réservoir et donc sa capacité à différencier deux entrées différentes. Si ce rayon est trop grand le réseau n'arrivera pas à "généraliser". Pour Mantas Lukoševičius [3], un grand rayon spectral est plus approprié pour des tâches nécessitant une mémoire à long terme des entrées.

5.2 Ajout d'apprentissage par plasticité intrinsèque

Une fois le code pris en main, il a fallu comprendre puis implémenter une règle d'apprentissage non-supervisé inspirée de la PI biologique. Cette règle d'apprentissage a été appliquée au réservoir de l'ESN : elle permettait au réseau de s'adapter à des hyper-paramètres peu optimisés. L'objectif était d'avoir un ESN performant sans passer par une trop longue période d'optimisation des hyper-paramètres. Les sections suivantes introduisent le concept de plasticité intrinsèque puis en décrivent l'implémentation pour un ESN.

5.2.1 La plasticité intrinsèque : définition

La plasticité intrinsèque (*intrinsic plasticity*) est un terme décrivant la capacité d'un neurone biologique à modifier son excitabilité selon la distribution des stimuli auxquels il est exposé. Plusieurs hypothèses tentent d'expliquer ce phénomène, l'une d'entre elles avance que l'objectif de la PI est de maximiser la transmission d'informations pour une moyenne d'activité du neurone donnée. Cette hypothèse se base sur la théorie de l'information de Shannon, montrant qu'une variabilité optimale des activations du neurone maximise la transmission d'informations. En d'autres termes, plus un neurone a des activations variées, plus il donne de l'information. Il a été observé par mesures in-vitro que les neurones du cortex visuel maintiennent une distribution exponentielle de leurs activations [9]. Cette expérience plaide en faveur de l'hypothèse précedemment énoncée, en effet les travaux de Stemmler et Koch [6] montrent qu'une distribution exponentielle maximise l'information pour une moyenne fixée.

5.2.2 La plasticité intrinsèque : implémentation

Les deux modélisations artificielles de la PI dans un RNA qui seront étudiées dans ce rapport sont celles décrites dans les travaux de Jochen J. Steil (deux articles publiés en 2007 [4] et 2008 [5]). La PI est modélisée par la modification de deux valeurs : le gain a et le biais b qui se combinent localement aux vecteurs d'entrée des neurones du réservoir. On a alors l'équation de mise à jour des neurones du réservoir suivante

$$x^{'}(n) = f(a(W^{in}[1; u(n)] + Wx(n-1)) + b), \tag{5}$$

où f représente la fonction tanh ou la fonction d'activation de Fermi (voir figure 7).

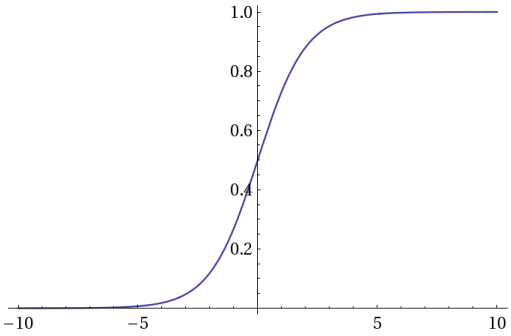


FIGURE 7 – Fonction de Fermi utilisé comme fonction d'activation des neurones du réservoir.

Les paramètres a et b de chaque neurone du réservoir sont mis à jour *en ligne* c'est à dire après chaque pas de temps du jeu d'entrainement.

Pour le cas des neurones du réservoir utilisant la fonction d'activation tanh on a les équations de mise à jour du gain et du biais suivantes :

$$\Delta b = -\eta(-\frac{\mu}{\sigma^2} + \frac{y}{\sigma^2}(2\sigma^2 + 1 - y^2 + \mu y)) \tag{6}$$

$$\Delta a = \frac{\eta}{a} + \Delta b x \tag{7}$$

avec η le ratio d'apprentissage, μ la moyenne théorique souhaitée pour la distribution, σ l'écart type théorique souhaité pour la distribution, x et y l'entrée et la sortie correspondante du neurone considéré. Pour les neurones utilisant tanh c'est la distribution gaussienne (figure 8) qui est recherchée,

c'est elle qui maximise le plus la transmission d'information pour ce type de neurones.

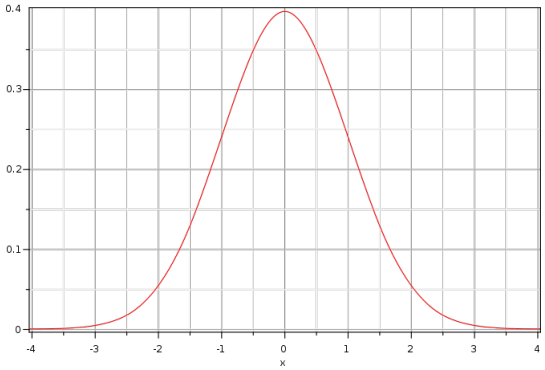


FIGURE 8 – Distribution gaussienne recherchée pour maximiser la transmission d'information des neurones du réservoir utilisant la fonction d'activation tanh. Sur ce graphique la moyenne est nulle et l'écart type est fixé à 1.

Pour le cas des neurones utilisant la fonction de Fermi on a les équations suivantes :

$$\Delta b = \eta(1 - (2 + \frac{1}{\mu})y + \frac{y^2}{\mu}) \tag{8}$$

$$\Delta a = \frac{\eta}{a} + \Delta b x \tag{9}$$

Pour cette fonction d'activation c'est la distribution exponentielle (figure 9) qui maximise la transmission d'information.

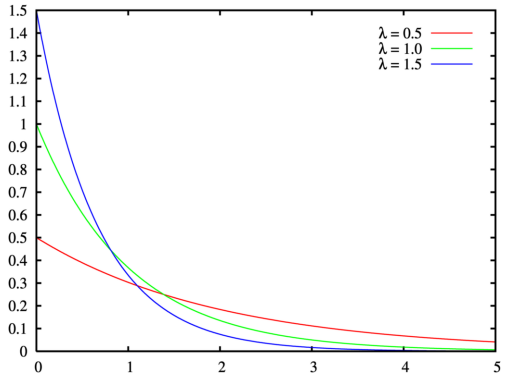


FIGURE 9 – Distribution exponentielle recherchée pour maximiser la transmission d'information des neurones du réservoir utilisant la fonction d'activation de Fermi. Différentes moyennes λ sont exposées dans ce graphique.

5.3 Prédiction de série chaotique

L'objectif était d'approximer la série chaotique temporelle de Mackey-Glass (voir figure 10). Pour ce problème, l'ensemble d'entrée $u(n)$ et de sortie $y^{target}(n)$ sont tous deux des nombres réels, d'où $N_u = N_y = 1$. Au départ la fonction d'activation des neurones était la tangente hyperbolique (voir figure 11)

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{10}$$

mais une option a été ajoutée pour pouvoir aussi utiliser la fonction d'activation de Fermi (figure 7).

$$f(x) = \frac{1}{1 + exp^{-x}} \tag{11}$$

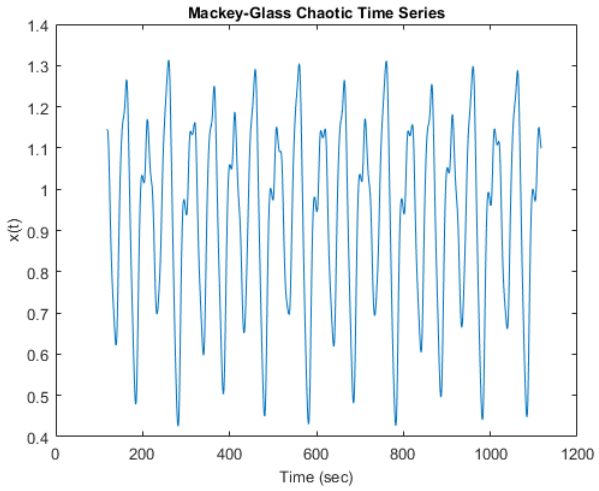


FIGURE 10 – Tâche de prédiction du réseau : approximer la série chaotique temporelle de Mackey-Glass.

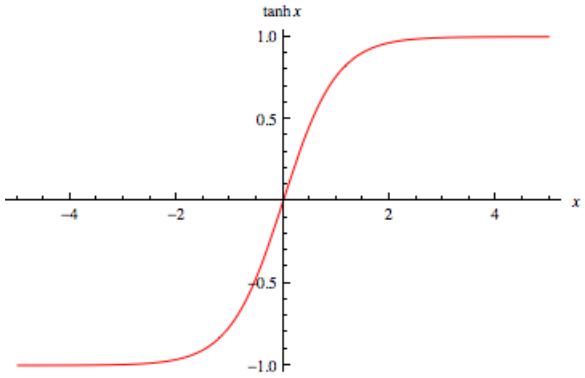


FIGURE 11 – Fonction d'activation tanh utilisé comme fonction d'activation des neurones du réservoir.

Les jeux d'entrainement et de test sont tout deux constitués de 2000 pas de temps. Le réservoir est initialisé durant la première époque (1 époque correspond au passage complet du jeu d'entrainement). La PI du réservoir est entraînée jusqu'à que le gain et le biais convergent, puis la regression linaire est effectuée.

5.4 Apprentissage de la grammaire d'Elman

L'ESN donné en début de stage puis modifié au cours de celui-ci a servi de base à l'implémentation. Il a fallu passer d'un ESN effectuant une régression à un ESN réalisant une tâche de catégorisation multiple. La dimension d'entrée et de sortie de ce nouveau réseau est donc $N_u = N_y = N$, avec N le nombre de mots de la grammaire. Une entrée du réseau est donc un mot (le caractère '.' est considéré comme un mot, *boy* et *boys* sont deux mots distincts) représentés par un vecteur binaire dont toutes les valeurs sont à 0 sauf l'indice équivalent au mot. Par exemple le mot *boy* est représenté par un vecteur de taille N, ici 24, dont tous les éléments sont à 0 sauf le premier qui est à 1.

Les jeux de test et d'entrainement, basés sur les travaux d'Arnaud Rachez [10], sont composés tous deux de 35000 mots : ils contiennent des phrases de longueurs variables utilisant la grammaire d'Elman, à l'exception du verbe *to chase* remplacé par *to hit*.

Table 1
Grammar used for training and test sets

S→NP VP "."
NP → PropN | N | NRC
VP → V | VNP
RC → who NP V | who VP
N → boy | girl | cat | dog | boys | girls | cats | dogs
PropN → john | mary
V → chase | feed | see | hear | walk | live | chases | feeds | sees | hears | walks | lives
With these additional restrictions:

- *Number agreement between N & V within a relative clause and between the head N and subordinate V.*
- *Verb arguments*:
  - *Hit, feed: require direct object (VP→V NP)*
  - *See, hear: optionally allow direct object (VP → V | VNP)*
  - *Walk, live: preclude direct object (VP→V)*
- *In the rule RC→who NP V, the Verb in the V must allow a direct object*

Pour initialiser les neurones réservoir, le jeu d'entrainement est donné au réseau une première fois (pour que celui-ci oublie son état initial), l'apprentissage de la PI est effectuée à partir de la seconde époque.

Les performances du réseau sont observables grâce à l'implémentation d'une des quatre métriques utilisés par Matthew H. Tong : le ratio d'accord nom/verbe.

$$\text{agreement rate} = \frac{\text{nb verbes accordés}}{\text{nb verbes total}} \tag{12}$$

Par exemple pour le sujet "Mary", "sees" est valide tandis que "see" ne l'est pas. Pour que le réseau accorde correctement un verbe à son sujet il faut que la moyenne de tous les indices de la prédiction de sortie correspondant aux verbes bien accordés soit supérieure à celle des mal accordés.

6 Résultats

Sachant que le stage ayant conduit à la rédaction de ce rapport a été court (1 mois) et que je n'ai jamais travaillé sur des RNAs auparavant, les résultats exposés dans cette section sont "à pondérer".

Le tableau suivant 6 résume les paramètres utilisés pour la création des deux ESNs et pour la plasticité intrinsèque lors des expériences réalisées. Par manque de temps aucune optimisation des paramètres n'a été effectuée.

	Prédiction série chaotique de Mackey-Glass avec PI	Apprentissage grammaire d'Elman (avec et sans PI)
Taille du réservoir	300	50 ou 300
Fonction d'activation des neurones du réservoir	tanh ou fermi	tanh
Ratio de fuite des neurones du réservoir	0.3	1 (pas de fuite)
Rayon spectral	1	0.98
Échelonnage de l'entrée (input scaling)	1	1.2
Coefficient de régularisation	0.02	0.02
PI : Ratio d'apprentissage	0.001	Si PI : 0.001
PI : moyenne théorique	tanh : 0 fermi : 0.2	Si PI : 0
PI : écart type théorique	tanh : 0.2 (non utilisé pour neurones fermi)	Si PI : 0.2

6.1 Résultats pour la prédiction de série chaotique

6.1.1 Ajout de plasticité intrinsèque

Après l'implémentation de la PI, des graphiques inspirés de ceux présentés dans les travaux de Jochen J.Steil ont été mis en place pour s'assurer d'avoir réussi à implémenter correctement la PI pour les deux fonctions d'activation. Pour chaque époque tout le jeu d'entrainement (de taille 2000) est donné au réservoir. La règle de PI est appliquée à chaque pas de temps sauf durant la première époque, dans un souci de comparaison avec la version sans PI. Après la dernière époque d'entraînement par PI les poids de la matrice de sortie W^{out} sont entrainés.

On peut observer dans la figure 12 que sans PI, le neurone choisi aléatoirement a des activations peu variées, elles sont contractées autour de 0,55.

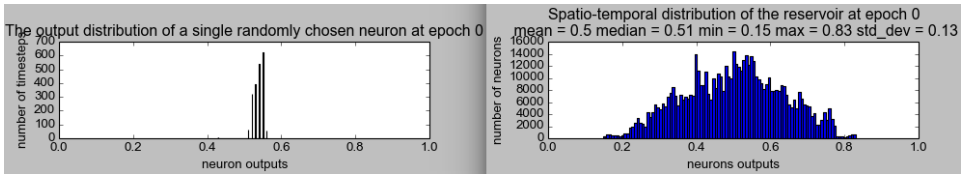


FIGURE 12 – Époque 0, pas d'entrainement par plasticité intrinsèque, la distribution des activations du neurone choisi aléatoirement est très contractée. Graphique de gauche : toutes les activations d'un neurone du réservoir au cours de l'époque 0 (soit 2000 pas de temps). Graphique de droite : Les activations de tous les neurones du réservoir au cours de l'époque 0.

Après 20 époques d'entraînement par PI (figure 13) les activations du réservoir approchent la moyenne théorique souhaitée de 0,2. La distribution se rapproche de la distribution exponentielle.

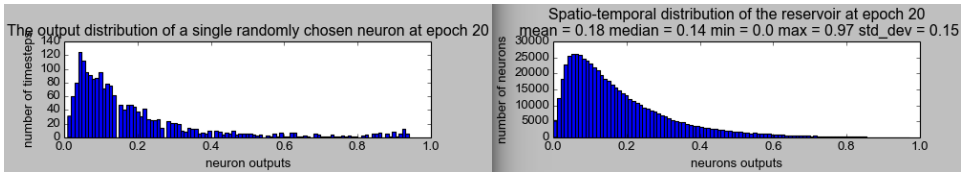


FIGURE 13 – Époque 20, la distribution spatio-temporelle globale au réseau et la distribution locale au neurone aléatoirement choisi tendent vers une distribution exponentielle. On peut remarquer que la moyenne s'est rapprochée de sa valeur théoriquement souhaitée : 0.2.

Après 40 époques d'entraînement (figure 14) la distribution exponentielle est clairement discernable, la moyenne de la distribution est de 0,19. On peut observer (figure 15) que le gain et le biais de la PI ont réussi à converger. Le gain suit approximativement une fonction racine carrée tandis que le biais converge vers -2.

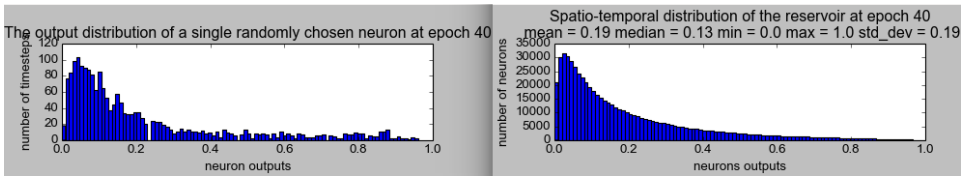


FIGURE 14 – Époque 40, la distribution exponentielle est clairement discernable.

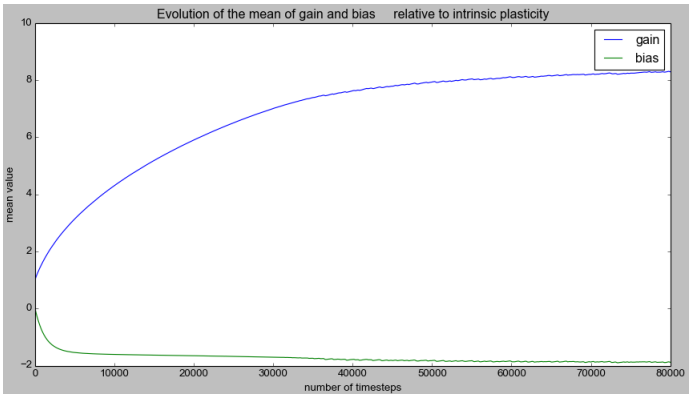


FIGURE 15 – Évolution de la moyenne du gain et du biais pour les neurones à la fonction de Fermi.

La PI a aussi été implémentée avec succès pour les neurones tanh. La moyenne a été fixée à 0 et l'écart type à 0.2. On peut observer que les résultats expérimentaux ont bien approché ces valeurs.

Après une époque passée sans entraînement (figure 16), on peut voir que les activations du neurones sont contractées. La distribution globale des sorties du réservoir n'a pas de forme particulière.

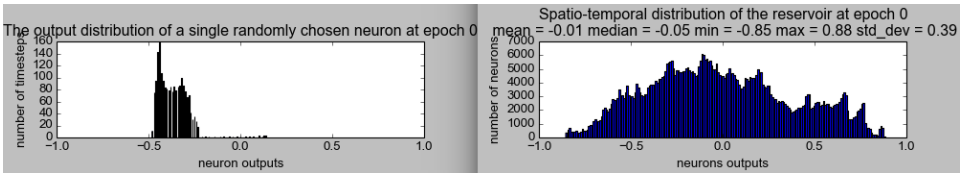


FIGURE 16 – Époque 0, pas d'entrainement par plasticité intrinsèque, la distribution des activations du neurone choisi aléatoirement est contractée autour de -0.4.

Après une première époque d'entraînement la moyenne et l'écart type théorique est bien approché (figure 17).La convergeance du gain et du bias (figure 19) est bien plus rapide avec tanh qu'avec les neurones utilisant la fonction de Fermi. Environ 1000 pas de temps suffisent pour faire converger le gain et le biais, de fait la deuxième époque où l'entraînement par PI est appliqué (figure 18) n'apporte que peu de changement. On observe que pour les neurones tanh le biais ne varie pas. Le gain approche une fois de plus une fonction racine carrée.

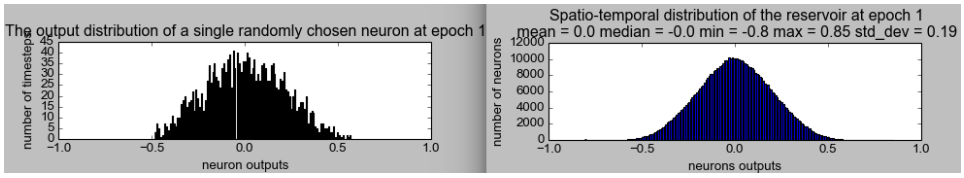


FIGURE 17 – Époque 1, la distribution spatio-temporelle globale au réseau et la distribution locale au neurone aléatoirement choisi tendent vers une distribution gaussienne. On peut remarquer que la moyenne et l'écart type ce sont rapprochés de leurs valeurs théoriquement souhaités : respectivement 0 et 0.2.

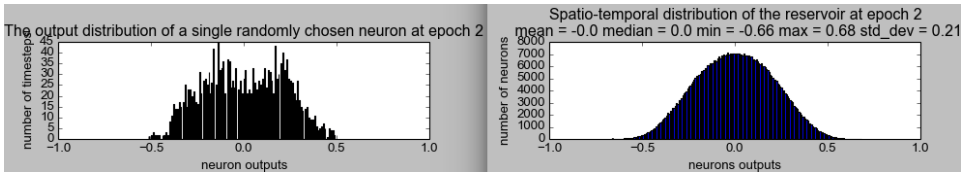


FIGURE 18 – Époque 2, la distribution gaussienne est clairement discernable. Peu de changement par rapport à la figure 17 car le gain s'est stabilisé dès 1000 pas de temps (voir figure 19).

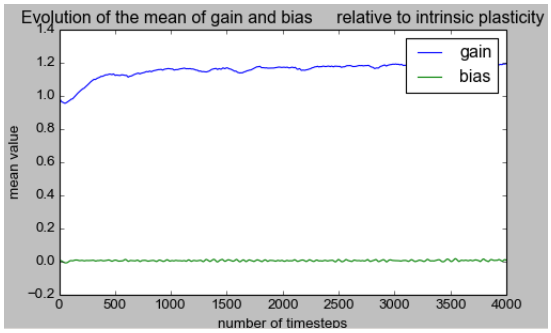


FIGURE 19 – Évolution de la moyenne du gain et du biais pour les neurones utilisant la fonction tangente hyperbolique. 1000 pas de temps suffisent pour que la moyenne du gain converge. Le biais ne varie pas.

Dans ses deux articles, Jochen.J.Steil [4] [5] montre que son ESN obtient de meilleurs résultats lorsqu'il utilise l'apprentissage par PI. Néanmoins, pour notre modèle, l'ESN simpliste donné en début de stage reste plus performant. En effet, compte tenu de la durée du stage, je n'ai que très peu optimisé l'ESN utilisant la PI, le but était plutôt de comprendre puis d'ajouter cette forme

d'apprentissage. Nous utilisons donc en majeure partie les hyper-paramètres qui furent implémentés par Mantas Lukoševičius, optimisés pour sa version de l'ESN.

6.1.2 Adaptation du réseau à la dégradation de son rayon spectral

Par la suite j'ai cherché à faire varier le rayon spectral du réservoir de l'ESN pour en étudier les conséquences sur la PI. En comparant les performances du réseau suivant le rayon spectral utilisé et si la PI était appliquée ou non j'ai pu observer que le réseau, grâce à la PI dans le réservoir, s'adaptait à la dégradation de son rayon spectral et maintenait plus ou moins ses performances. Pour cette expérience les neurones du réservoir on la fonction d'activation tanh. La figure 20 montre l'évolution du gain et du biais avec un rayon spectral à 1.

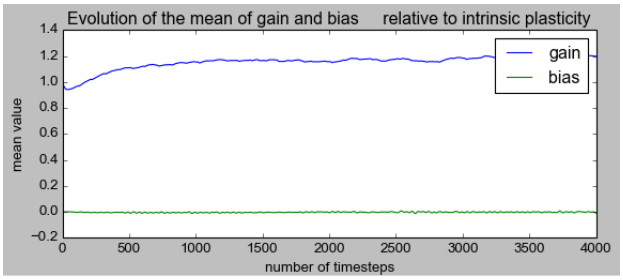


FIGURE 20 – Evolution du gain et du biais pour un rayon spectral = 1 après 2 époques d'entraînement par PI.

Si on retire $0,5$ à ce rayon spectral la convergeance du gain et du biais s'en retrouve changé (voir figure 21). La courbe du gain qui convergeait vers $1,2$ en 1000 pas de temps a désormais besoin d'environ 8000 pas de temps (4 époques) pour converger vers $2,5$. Plus on diminue le rayon spectral plus il faudra de temps au gain pour converger.

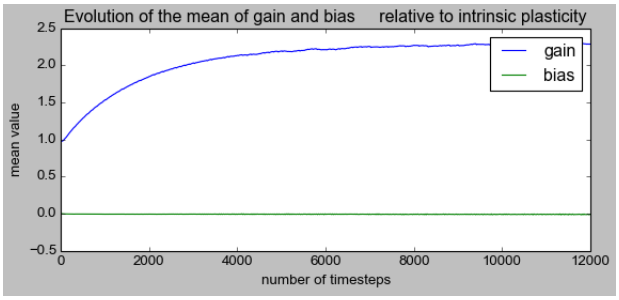


FIGURE 21 – Evolution du gain et du biais pour un rayon spectral = 0.5 après 6 époques d'entraînement par PI. La diminution du rayon spectral entraîne une augmentation du temps de convergeance du gain.

Lorsqu'on augmente le rayon spectral à $1,5$ (voir figure 22) le gain chute de 1 à environ 0.8 en une centaine de pas de temps puis augmente légerement pour converger jusqu'à $0,85$. Plus on augmente le rayon spectral, plus la chûte du gain est élevée.

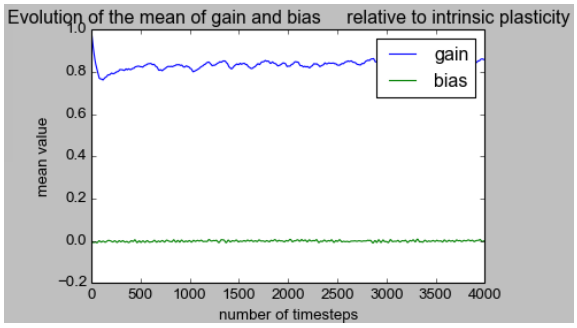


FIGURE 22 – Evolution du gain et du biais pour un rayon spectral = 1.5 durant 2 époques d'entraînement par PI. L'augmentation du rayon spectral accentue la chute de la moyenne du gain durant les premiers pas de temps.

Les performances du réseau restent similaires lorsqu'on modifie le rayon spectral et qu'on utilise la PI (augmentation de la RMSE d'un facteur 10 pour un ajout de 0.5 au rayon spectral). Néanmoins sans PI les performances du réseau diminuent fortement (augmentation de la RMSE d'un facteur 1000 à 10000) lorsqu'on ajoute $0,5$ au rayon spectral. Ces observations permettent de mettre en évidence la capacité de l'apprentissage par PI de compenser un rayon spectral peu optimal.

6.2 Résultats pour l'apprentissage de la grammaire d'Elman

6.2.1 Performances de l'ESN

Les tests ont permis d'observer que l'ESN réussit à apprendre relativement bien la grammaire. Une fois les poids de la matrice de sortie entraînés, il parvient à atteindre en moyenne (sur 5 ESNs aux poids générés aléatoirement) 76% et 83% de précision au niveau de l'accord nom/verbe pour un réservoir de 50 et 300 neurones, respectivement.

Pour mieux visualiser les prédictions du réseau j'ai implémenté des graphiques (voir figure 23) inspirés de ceux de Matthew H. Tong. Cet ensemble de courbes illustre les activations du réseau pour la prédiction de chaque mots de la phrase "boy who dog feeds hears ." . L'objectif était d'approcher pour chaque prédiction de mot les probabilités réelles que tel ou tel mot soit le suivant. Les activations de mon ESN sont bien loin de cet objectif (valeur au dessus de 1 ou bien négative, somme des valeurs très éloignée de 1) mais pour chaque prédiction on peut clairement voir quel mot ou classe de mots est prédit par le réseau.

On peut observer que le réseau fait une erreur de prédiction pour "dog" cependant l'erreur n'est pas totale : sa plus forte prédiction est un verbe accordé pour la troisième personne du singulier, ce qui aurait pu être possible car une phrase du type "boy who walks ." est valide. Dans son article Matthew H. Tong [8] déclare atteindre 89% et 92% (pour un réservoir de 50 et 300 neurones) d'accords nom/verbe correct. La différence de performance avec l'ESN de Matthew H. Tong réside dans le fait que l'article ne communique pas tous les hyper-paramètres de leur ESN, je me suis donc inspiré des précédents travaux de prédiction de série chaotique pour choisir les paramètres manquants, peu optimisés pour l'apprentissage de la grammaire d'Elman.

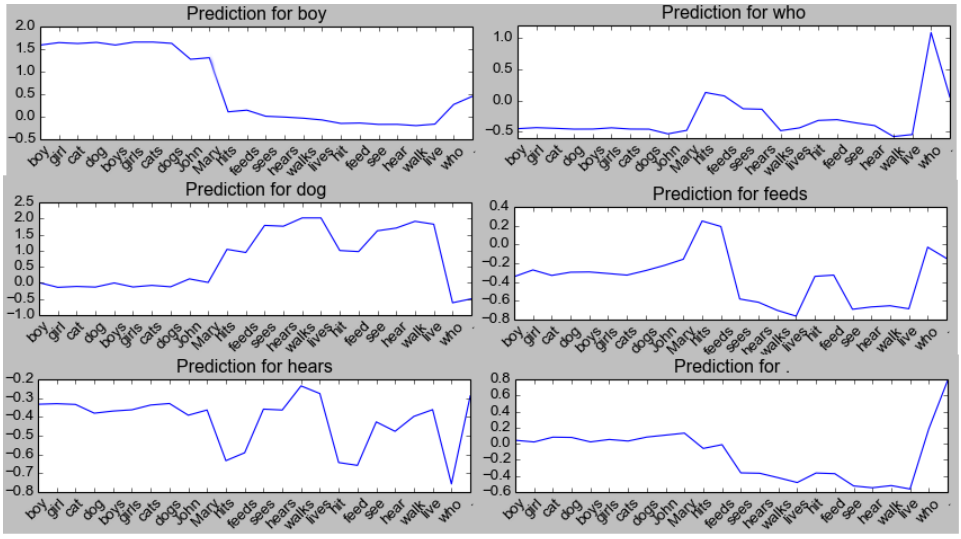


FIGURE 23 – Prédictions données par le réseau pour la phrase "boy who dog feeds hears .". Moyenne des activations du vecteur de sortie après 5 exécutions sur des ESNs ayant un réservoir de 300 neurones dont les poids ont été générés aléatoirement.

6.2.2 Ajout de plasticité intrinsèque

L'ajout de plasticité intrinsèque s'est révélé être bénéfique pour le réseau, du moins en terme de pourcentage de verbes accordés. La convergeance du gain (voir figure 24) est effectué en 1000 pas de temps et le biais, comme dans les expériences précédentes, ne varie pas. Le réseau atteint 82% et 84% de précision au niveau de l'accord nom/verbe pour un réservoir de 50 et 300 neurones, respectivement. On observe que plus le réservoir est grand, plus l'écart entre les performances du réseau avec ou sans PI se réduit. Le gain de performance est significatif pour un petit réservoir.

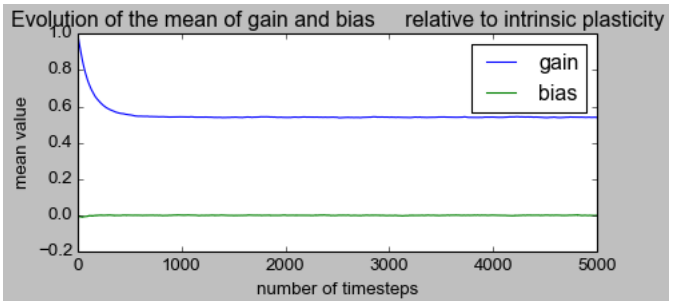


FIGURE 24 – Évolution du gain et du biais de la plasticité intrinsèque du réservoir durant l'apprentissage de la grammaire d'Elman. La convergence du gain nécéssite environ 1000 pas de temps.

7 Bilan

7.1 Conclusion sur les résultats et idées de futures recherches

Durant ce stage d'un mois, j'ai utilisé un ESN minimaliste, implémenté par Mantas Lukoševičius, auquel j'ai ajouté une méthode d'apprentissage non-supervisé dans le réservoir. Nommée plasticité intrinsèque, l'implémentation de cette méthode bio-inspirée se base sur deux articles de Jochen J. Steil [4] [5]. J'ai pu mettre en évidence que la PI appliquée à des neurones tanh converge bien plus vite qu'avec des neurones fermi. La PI permet aussi de compenser un rayon spectral peu optimal. Il pourrait être interessant d'étudier plus en profondeur le rapport entre la PI et les autres hyper-paramètres du réservoir.

Un ESN apprenant la grammaire d'Elman, basé sur l'article de Matthew H. Tong [8], a aussi été implémenté au cours du stage. L'ajout de plasticité intrinsèque c'est révélé bénéfique au niveau des performances. Des recherches complémentaires, sur différentes grammaires, s'avèrent néanmoins nécéssaires pour s'assurer que la PI augmente bel et bien les performances d'un ESN sur ce type de tâches. En effet, étant donné la courte durée de la période d'expérimentation ainsi que de mon niveau d'expertise limité, ces résultats restent à confirmer par une exploration plus exhaustive du sujet.

7.2 Bilan personnel du stage

Ce stage fût un réel défi pour moi car le sujet était difficile, l'ESN est un réseau de neurones artificiels délicat à prendre en main. J'ai pu découvrir le monde de la recherche en informatique, qui me plaît beaucoup et me conforte

dans mon projet professionnel : devenir enseignant-chercheur. Ce fût l'occasion pour moi d'être initié à l'apprentissage automatique, un de mes thèmes préférés en informatique, mais aussi de découvrir ses applications en neurosciences. Durant ce stage, j'ai appris à me documenter grâce à des articles scientifiques spécialisés. J'ai étoffé mes connaissances en Python, LaTeX et en calculs matriciels. La rédaction de ce rapport mais aussi la présentation orale des expériences que j'ai réalisée devant l'équipe *Mnemosyne* m'ont permis d'apprendre à rendre compte de mon travail.

8 Annexes

8.1 ESN utilisé pour prédire la série chaotique de Mackey-Glass

8.1.1 Détails d'implémentation

Les données de test et d'entraînement de cet ESN ont été tirées du site web de Mantas Lukoševičius `http://minds.jacobs-university.de/mantas/code`. Au départ j'ai commencé à travailler sur l'ESN minimaliste de Mantas Lukoševičius auquel Xavier Hinaut, mon maître de stage, avait fait des modifications. J'ai ajouté plusieurs modes :

activation_function_mode permet de choisir la fonction d'activation désirée pour les neurones du réservoir : tanh ou fermi.

wout_mode permet d'ajouter ou non le signal d'entrée du réseau à l'entrée de la couche de sortie.

ip_mode permet de choisir si on applique ou non la PI dans le réservoir.

ip_update_mode permet de choisir ce qui est considéré comme la sortie du neurone dans les équations de mise à jour du gain et du biais. Cette sortie sera uniquement la première partie de l'équation de mise à jour du neurone (2) ou alors l'intégralité de la mise à jour en prenant en compte la fuite du neurone ((2) et (3)).

8.1.2 Code source

```
"""
A minimalistic Echo State Networks demo with Mackey-Glass (delay 17) data
in "plain" scientific Python.
by Mantas Lukosevicius 2012
http://minds.jacobs-university.de/mantas
---
Modified by Xavier Hinaut: 19 November 2015.
http://www.xavierhinaut.com

Modified by Remy Portelas: 30 May 2016
http://www.traineeshavenowebsites.com
"""
from numpy import *
from matplotlib.pyplot import *
import scipy.linalg

def set_seed(seed=None):
    """Making the seed (for random values) variable if None"""

    # Set the seed
    if seed is None:
        import time
        seed = int((time.time()*10**6) % 10**12)

    try:
        random.seed(seed) #np.random.seed(seed)
        print "Seed used for random values:", seed
    except:
        print "!!! WARNING !!!: Seed was not set correctly."
    return seed

#reservoir's neurons activation function
def sigmoid(x):
    if activation_function_mode == 'tanh':
        return tanh(x)
    elif activation_function_mode == 'fermi':
        return ( 1 / ( 1 + exp(-x)))
    else:
        raise Exception, "ERROR: 'activation_function_mode' was not " + \
        "set correctly."

#plot the activation of all neurons in the reservoir during 1 epoch
def plot_activity(x,epoch):
    figure(epoch+42).clear()
    if activation_function_mode == 'tanh':
         hist(x.ravel(), bins = 200)
         xlim(-1,+1)
```

```
    else :#fermi
        hist(x.ravel(), bins = 100)
        xlim(0,+1)
    xlabel('neurons outputs')
    ylabel('number of neurons')
    #compute some caracteristics of the distribution
    mean = str(round(x.mean(), 2))
    med = str(round(median(x), 2))
    min = str(round(x.min(), 2))
    max = str(round(x.max(), 2))
    std_dev = str(round(x.std(), 2))
    title('Spatio-temporal distribution of the reservoir at epoch ' + \
    str(epoch) + '\n mean = '+ mean + ' median = ' + med + ' min = ' + \
    min + ' max = ' + max + ' std_dev = ' + std_dev)

def plot_neuron_activity(x,epoch):
    figure(epoch+84).clear()
    if activation_function_mode == 'tanh':
        hist(x.ravel(), bins = 200)
        xlim(-1,+1)
    else :#fermi
        hist(x.ravel(), bins = 100)
        xlim(0,+1)
    xlabel('neuron outputs')
    ylabel('number of timesteps')
    #compute some caracteristics of the distribution
    title('The output distribution of a single randomly chosen neuron ' + \
    'at epoch ' + str(epoch))

# load the data
trainLen = 2000
testLen = 2000

data = loadtxt('MackeyGlass_t17.txt')

# plot some of it
figure(10).clear()
plot(data[0:1000])
title('A sample of data')

mode = 'prediction'  #given x try to predict x+1
#mode = 'generative' #compute x and use it as an input to compute x+1

activation_function_mode = 'tanh'
#activation_function_mode = 'fermi'

wout_mode = 'entries bias and resOut'
#wout_mode = 'resOut and bias only'
```

```
ip_mode = 'intrinsic_plasticity_on'
#ip_mode = 'intrinsic plasticity off'

#IP parameter
#ip_update_mode = 'leaky neurons treated'
ip_update_mode = 'leaky_neurons_ignored'

#Set the number of training's epochs,
if ip_mode == 'intrinsic_plasticity_on':
  if activation_function_mode == 'tanh':
      nb_epoch = 3 #IP with tanh neurons needs less time to converge
  else:
    nb_epoch = 41
else: #if no IP, then we don't need multiple epochs of training
  nb_epoch = 1

# generate the ESN reservoir
inSize = outSize = 1 #input/output dimension
resSize = 300 #reservoir size
a = 0.3 # leaking rate
if ip_mode == 'intrinsic_plasticity_on':
    spectral_radius = 1.
    reg = 0.02  #regularization coefficient
    #init Intrisic Plasticity (IP)
    lr = 0.001  #learning rate
    if activation_function_mode == 'tanh':
        m = 0.       #mean
        sigma = 0.2 #standard deviation 0.2 gives best results
        var = square(sigma) #variance

    else : #fermi
        m = 0.2
    #instanciate some matrix to store the evolution of IP's gain and bias
    ip_gain = ones((resSize, 1))
    record_ip_gain = zeros(((nb_epoch-1) * trainLen , 1))
    record_ip_bias = zeros(((nb_epoch-1) * trainLen , 1))
    ip_bias = zeros((resSize, 1))


else :          #IP off
    spectral_radius = 1.25
    reg = 1e-8 # regularization coefficient

input_scaling = 1.

#change the seed, reservoir performances should be averaged accross at least
#20 random instances (with the same set of parameters)
our_seed = None #Choose a seed or None
set_seed(our_seed)
```

```
#generation of random weights
Win = (random.rand(resSize,1+inSize)-0.5) * input_scaling
W = random.rand(resSize,resSize)-0.5

# Option 1 - direct scaling (quick&dirty, reservoir-specific):
#W *= 0.135
# Option 2 - normalizing and setting spectral radius (correct, slow):
print 'Computing spectral radius...',
rhoW = max(abs(linalg.eig(W)[0])) #maximal eigenvalue
print 'done.'
W *= spectral_radius / rhoW

# allocated memory for the design (collected states) matrix
if wout_mode == 'entries bias and resOut':
    X = zeros((1+inSize+resSize,trainLen))

elif wout_mode == 'resOut and bias only':
    X = zeros((1+resSize,trainLen))
else :
        raise Exception, "ERROR: 'wout_mode' was not set correctly."

#to display the spatio-temporal activity of an entire epoch
recorded_res_out = zeros((trainLen, resSize))

#choose a random neuron in the res to create a histogram of its activations
choosen_neuron = random.random_integers(resSize - 1)
neuron_out_records = np.zeros(trainLen)

# set the corresponding target matrix directly
Yt = data[None,1:trainLen+1]

# run the reservoir with the data and collect X
x = zeros((resSize,1))
for epoch in range(nb_epoch):
    for t in range(trainLen):
        u = data[t]
        res_in = dot( Win, vstack((1,u))) + dot( W, x )
        #compute reservoir activations with or without IP
        if ip_mode == 'intrinsic plasticity on':
            res_out = sigmoid( ip_gain * res_in + ip_bias )
            x = (1-a) * x + a * res_out
            #compute delta_bias considering the activation function
            #we don't want to train our network during the first epoch
            if epoch != 0:
                if activation_function_mode == 'tanh':
                    if ip_update_mode == 'leaky neurons ignored':
                        d_ip_bias = (-lr) * ((-(m / var)) + (res_out / var) * \
                        ((2 * var) + 1 - square(res_out) + m * res_out))
```

```
                elif ip_update_mode == 'leaky_neurons_treated':
                    d_ip_bias = (-lr) * ((-(m / var)) + (x / var ) * \
                    ((2 * var) + 1 - square(x) + m * x))
                else:
                    raise Exception, "ERROR: 'ip_update_mode' was not " \
                    "set correctly."
            else: #fermi
                if ip_update_mode == 'leaky_neurons_ignored':
                    d_ip_bias = lr * (1 - (2 + (1/m)) * res_out + \
                    (square(res_out) / m))
                elif ip_update_mode == 'leaky_neurons_treated':
                    d_ip_bias = lr * (1 - (2 + (1/m)) * x + \
                    (square(x) / m))
                else:
                    raise Exception, "ERROR: 'ip_update_mode' was not " \
                    "set correctly."
            #compute delta_bias and update IP's gain and bias
            ip_bias += d_ip_bias
            ip_gain += (lr / ip_gain) + (d_ip_bias * res_in)
            #store the results to plot them
            record_ip_bias[t + (trainLen * (epoch-1)),0] = ip_bias.mean()
            record_ip_gain[t + (trainLen * (epoch-1)),0] = ip_gain.mean()


    elif ip_mode == 'intrinsic_plasticity_off':
        res_out = sigmoid(res_in)
        x = (1-a) * x + a * res_out

    else:
       raise Exception, "ERROR: 'ip_mode' was not set correctly."
    #accumulate values of a randomly choosen reservoir's neuron activations
    neuron_out_records[t] = round(res_out[choosen_neuron],2)

    #we perform linear regression after the last epoch of training
    #so we only store the activations of the last epoch
    if epoch == nb_epoch - 1 :
        if wout_mode == 'entries_bias_and_resOut':
            X[:,t] = vstack((1,u,x))[:,0]

        elif wout_mode == 'resOut_and_bias_only':
            X[:,t] = vstack((1,x))[:,0]

        else :
            raise Exception, "ERROR: 'wout_mode' was not set correctly."
    #store spatio-temporal activity of the reservoir
    recorded_res_out[t] = res_out[:,0]

#plot some signals to see if IP works
if activation_function_mode == 'tanh':
```

```
            plot_activity(recorded_res_out, epoch)
            plot_neuron_activity(neuron_out_records, epoch)
        if activation_function_mode == 'fermi':
            if(epoch%20 == 0):
                plot_activity(recorded_res_out, epoch)
                plot_neuron_activity(neuron_out_records, epoch)

        # plot the evolution of gain and bias during training
        if ip_mode == 'intrinsic plasticity on':
            figure(10).clear()
            plot( record_ip_gain, label='gain' )
            plot( record_ip_bias, label='bias' )
            legend()
            ylabel('mean value')
            xlabel('number of timesteps')
            title('Evolution of the mean of gain and bias ' \
            'relative to intrinsic plasticity')

    # train the output
    X_T = X.T
    # use ridge regression (linear regression with regularization)
    if wout_mode == 'entries bias and resOut':
        Wout = dot( dot(Yt,X_T), linalg.inv( dot(X,X_T) + \
        reg*eye(1+inSize+resSize)))

    elif wout_mode == 'resOut and bias only':
        Wout = dot( dot(Yt,X_T), linalg.inv( dot(X,X_T) + \
        reg*eye(1+resSize)))

    else :
        raise Exception, "ERROR: 'wout_mode' was not set correctly."

    # use pseudo inverse
    #Wout = dot( Yt, linalg.pinv(X) )

    # run the trained ESN with the test set
    Y = zeros((outSize,testLen))
    u = data[trainLen]

    for t in range(testLen):
        res_in = dot( Win, vstack((1,u)) ) + dot( W, x )
        if ip_mode == 'intrinsic plasticity on':
            res_out = sigmoid(ip_gain * res_in + ip_bias )

        elif ip_mode == 'intrinsic plasticity off':
           res_out = sigmoid(res_in)

        else:
           raise Exception, "ERROR: 'ip_mode' was not set correctly."
```

```
        x = (1-a) * x + a * res_out

        if wout_mode == 'entries bias and resOut':
            y = dot( Wout, vstack((1,u,x)) )

        elif wout_mode == 'resOut and bias only':
            y = dot( Wout, vstack((1,x)))

        else :
            raise Exception, "ERROR: 'wout_mode' was not set correctly."

        Y[:,t] = y
        if mode == 'generative':
            # generative mode:
            u = y
        elif mode == 'prediction':
            # predictive mode:
            u = data[trainLen+t+1]
        else:
            raise Exception, "ERROR: 'mode' was not set correctly."

# compute MSE for the first errorLen time steps
errorLen = 500
mse_for_each_t = square( data[trainLen+1:trainLen+errorLen+1] - \
Y[0,0:errorLen] )
mse = sum( mse_for_each_t ) / errorLen
print 'MSE = ' + str( mse )
print 'compared to max default (Mantas) error 2.91524629066e-07' \
'(For prediction / 100 Neurons)'
print 'ratio compared to (Mantas) error ' + str(mse/2.91524629066e-07) + \
'  (For prediction / 100 Neurons)'
print ""
print 'compared to max default (Mantas) error 4.06986845044e-06 ' \
'(For generation / 1000 Neurons)'
print 'compared to max default (Mantas) error 2.02529702465e-08 ' \
'(For prediction / 1000 Neurons)'
show()
```

8.1.3 ESN utilisé pour apprendre la grammaire d'Elman

8.1.4 Détails d'implémentation

Pour l'apprentissage de la grammaire d'Elman les données de test et d'entraînement se trouvent sur le dépot Github d'Arnaud Rachez [10]. Cependant ce sont les jeux utilisés par Elman [7], ils contiennent des phrases de longueur croissante. Pour me rapprocher des données utilisées par Matthew H. Tong j'ai effectué un mélange aléatoire des phrases dans chaque jeu. Ces nouveaux jeux ainsi créés restent tout de même éloignés de ceux de Matthew H. Tong (longueur moyenne des phrases et longueur totale des jeux différentes).

La variable **nb_network** permet de choisir le nombre d'ESNs qui seront générés aléatoirement. Le ratio d'accord nom/verbe et les graphique donnés en fin d'exécution du code résultent de la moyenne des performances de tous ces ESNs.

Les prédictions de chaque neurone de sortie du réseau, lors de la phase de test, sont normalisées pour obtenir une moyenne nulle et un écart type de 1. Ce procédé, nommé *Z-scoring*, est utilisé par Matthew H. Tong pour éviter des erreurs liées au mots 'who' et '.' qui sont plus fréquents et ont de larges intervalles d'activation. Ce procédé permet d'améliorer les prédictions du réseau, qui sont désormais mesurée par l'écart type à la moyenne au lieu de l'activation du réseau.

8.1.5 Code source

```
"""
Created on Mon May 23 13:26:46 2016
@author: Remy Portelas
http://www.traineeshavenowebsites.com
---
Based on the minimalistic Echo State Networks by Mantas Lukosevicius 2012
enhanced by  Xavier Hinaut.
in "plain" scientific Python.
"""
import numpy as np
from matplotlib.pyplot import *
import scipy.linalg
import cPickle as pickle

#take an array of target words and
#an array of prediction arrays to plot results
def plot_predictions(target_words, predictions):
    x = np.arange(class_nb)
    my_xticks = vocabulary

    for i in range(predictions.shape[0]):
        prediction = predictions[i,:]
        figure(i).clear()
        title('Prediction for ' + target_words[i])
        xticks(x, my_xticks, rotation = 45)

        if scale_mode == 'scale entries' :
            #unshift data by +0.5 for ease of visualisation
            prediction += 0.5
        plot(x, prediction)

#display the spatio temporal distribution of
#reservoir's neurons during 1000 timesteps
def plot_activity(x,epoch):
    figure(epoch+42).clear()
    hist(x.ravel(), bins = 200)
    xlim(-1,+1)
    xlabel('neurons outputs')
    ylabel('number of timesteps')
    #compute some caracteristics of the distribution
    mean = str(round(x.mean(), 2))
    med = str(round(np.median(x), 2))
    min = str(round(x.min(), 2))
    max = str(round(x.max(), 2))
    std_dev = str(round(x.std(), 2))
    title('Spatio-temporal distribution of the reservoir within 1000 timesteps'
```

```
    ' at epoch ' + str(epoch) + '\n mean = '+ mean + ' median = ' + med + \
    ' min = ' + min + ' max = ' + max + ' std_dev = ' + std_dev)

#display the activity of a neuron during 1000 timesteps
def plot_neuron_activity(x,epoch):
    figure(epoch+84).clear()
    hist(x.ravel(), bins = 200)
    xlim(-1,+1)
    xlabel('neuron outputs')
    ylabel('number of timesteps')
    #compute some caracteristics of the distribution
    title('The output distribution of a single randomly chosen neuron' \
    'at epoch ' + str(epoch))

def set_seed(seed=None):
    """Making the seed (for random values) variable if None"""

    # Set the seed
    if seed is None:
        import time as t
        seed = int((t.time()*10**6) % 10**12)

    try:
        np.random.seed(seed) #np.random.seed(seed)
        print "Seed used for random values:", seed
    except:
        print "!!! WARNING !!!: Seed was not set correctly."
    return seed

#binary matrix refer to an array of 0 and 1
#turn the list of x integer into a x*class_nb matrix
#the idea: 3 becomes [0,0,0,1,0,...] (class_nb size)
def int_list_to_shifted_binary_matrix(int_list, class_nb):
    res = np.zeros((int_list.size,class_nb))
    for i in range(int_list.size):
        res[i,int_list[i]] = 1

    if scale_mode == 'scale_entries' :
        #input and output are shifted by -0.5
        #to scale well with the tanh activation function
        return res -0.5
    elif scale_mode == 'no_scaling' :
        return res
    else :
        raise Exception, "ERROR: 'scale_mode' was not set correctly."

#given a vector of 0 and 1 (if no scaling) or -0.5 and 0.5 (if scaling),
#return the corresponding word according to the vocabulary
def compute_word(bin_vec):
```

```
    if scale_mode == 'scale entries' :
        return vocabulary[np.where(bin_vec == 0.5)[0][0]]
    elif scale_mode == 'no scaling' :
        return vocabulary[bin_vec.nonzero()[0][0]]
    else :
        raise Exception, "ERROR: 'scale_mode' was not set correctly."

#given a network output vector,
#return 'singular' is the mean of all singular verbs probability
#is higher than plural ones or return 'plural' if not.
def compute_class(out):
    mean_singular = 0
    mean_plural = 0
    for v in singular_verbs:
        mean_singular += out[vocabulary.index(v)]
    mean_singular = mean_singular / singular_verbs.__len__()
    for v in plural_verbs:
        mean_plural += out[vocabulary.index(v)]
    mean_plural = mean_plural / plural_verbs.__len__()
    if mean_singular > mean_plural :
        return 'singular'
    return 'plural'

ip_mode = 'intrinsic plasticity on'
#ip_mode = 'intrinsic plasticity off'

# if "scale entries" is active, the input will be shifted by -0.5 to scale
# with reservoir's tanh activation function
#scale_mode = 'scale entries'
scale_mode = 'no scaling'


#don't change this seed to generate the same dataset as in the report
data_shuffle_seed = 989898
set_seed(data_shuffle_seed)

#load the data
with open('t5_train') as f:
    text_train =(' '.join(pickle.load(f))).split(' . ')
    np.random.shuffle(text_train)
    text_train = (' . '.join(text_train)).split(' ')

with open('t5_test') as f:
    text_test =(' '.join(pickle.load(f))).split(' . ')
    np.random.shuffle(text_test)
    text_test = (' . '.join(text_test)).split(' ')
#the shuffle extract the last '.'
text_train.append('.')
```

```
#display a samble of the testing set
print text_test[0:10]

#init the list of possible words
vocabulary = ['boy', 'girl', 'cat', 'dog', 'boys', 'girls', 'cats',
              'dogs', 'John', 'Mary', 'hits', 'feeds', 'sees',
              'hears', 'walks', 'lives', 'hit', 'feed', 'see',
              'hear', 'walk', 'live', 'who', '.']

#instanciate classes of verbs
singular_verbs = ('sees','hears','hits','walks','lives','feeds')
plural_verbs = ('see','hear','hit','walk','live','feed')
verbs = singular_verbs + plural_verbs

#change words by they proper index
u_data_train = np.asarray([vocabulary.index(w) for w in text_train[:-1]])
y_data_train = np.asarray([vocabulary.index(w) for w in text_train[1:]])
u_data_test = np.asarray([vocabulary.index(w) for w in text_test[:-1]])
y_data_test = np.asarray([vocabulary.index(w) for w in text_test])

#create matrices to run the multi-classification network
class_nb = vocabulary.__len__()
u_train = int_list_to_shifted_binary_matrix(u_data_train,class_nb)
y_train = int_list_to_shifted_binary_matrix(y_data_train,class_nb)

u_test = int_list_to_shifted_binary_matrix(u_data_test,class_nb)
y_test = int_list_to_shifted_binary_matrix(y_data_test,class_nb)

#training and testing parameters
nb_network = 1 #nb_network will be trained and the performances
               #will be averaged across them
nb_epoch = 2
trainLen = u_train.shape[0]
testLen = u_test.shape[0]

# generate the ESN reservoir
inSize = outSize = class_nb #input/output dimension
resSize = 50 #reservoir size
a = 1 # leaking rate, (a = 1) <==> no leaky neurons
spectral_radius = 0.98
input_scaling = 1.2
reg = 0.02 # regularization coefficient

if (ip_mode == 'intrinsic_plasticity_on'):
     #initialisation of IP parameters
      lr = 0.001 #IP learning rate
     m = 0.      #will store the mean of the reservoir activations
      sigma = 0.2 #standard deviation 0.2 gives best results
      var = np.square(sigma) #variance
```

```
        #instanciate some matrix to store the evolution of IP's gain and bias
        ip_gain = np.ones((resSize, 1))
        record_ip_gain = np.zeros((5000 , 1))
        record_ip_bias = np.zeros((5000 , 1))
        ip_bias = np.zeros((resSize, 1))

#instanciate perfomances metrics
mse = 0
agreement_rate = 0
averaged_recorded_prediction = np.zeros((6,class_nb))
for network in range(nb_network) :
    #create a network with random weigths
    our_seed = None #42 * network
    set_seed(our_seed)
    Win = (np.random.rand(resSize,1+inSize)-0.5) * input_scaling
    W = np.random.rand(resSize,resSize)-0.5
    # Option 1 - direct scaling (quick&dirty, reservoir-specific):
    #W *= 0.135
    # Option 2 - normalizing and setting spectral radius (correct, slow):
    print 'Computing spectral radius...',
    rhoW = max(abs(np.linalg.eig(W)[0]))
    print 'done.'
    W *= spectral_radius / rhoW


    #to display the spatio-temporal activity of 1000 timesteps
    recorded_res_out = np.zeros((1000, resSize))

    #choose a random neuron in the res
    #to create a histogram of its activations
    choosen_neuron = np.random.random_integers(resSize - 1)
    neuron_out_records = np.zeros(1000)

    #IP init
    ip_gain = 1
    ip_bias = 0

    # allocated memory for the design (collected states) matrix
    X = np.zeros((1+inSize+resSize,trainLen))

    # set the corresponding target matrix directly
    Yt = y_train.T

    # run the reservoir with the data and collect X
    x = np.zeros((resSize,1))
    for epoch in range(nb_epoch):
        for t in range(trainLen):
            #load the new input
            u = np.reshape(u_train[t],(u_train[t].shape[0],1))
```

```
            res_in = np.dot( Win, np.vstack((1, u))) + np.dot( W, x )
            #compute reservoir activation
            if ip_mode == 'intrinsic_plasticity_on':
                res_out = np.tanh( ip_gain * res_in + ip_bias )
                if (epoch !=0) & (t<5000) :
                    d_ip_bias = (-lr) * ((-(m / var)) + (res_out / var ) * \
                    ((2 * var) + 1 - np.square(res_out) + m * res_out))
                    #compute delta_bias and update IP's gain and bias
                    ip_bias += d_ip_bias
                    ip_gain += (lr / ip_gain) + (d_ip_bias * res_in)
                    #store the results to plot them
                    record_ip_bias[t,0] += ip_bias.mean()
                    record_ip_gain[t,0] += ip_gain.mean()
            else :
                res_out = np.tanh(res_in)

            x = (1-a) * x + a * res_out
            #the first epoch initialise the network,
            #we start storing results and training IP during the second epoch
            if(trainLen-t) <= 1000 :
                #store spatio-temporal activity of the reservoir
                recorded_res_out[trainLen-t-1] = res_out[:,0]
                #accumulate values of a randomly choosen
                #reservoir's neuron activations
                neuron_out_records[trainLen-t-1] = round(res_out[choosen_neuron],2)

            #we perform linear regression after the last epoch of training
            #so we only store the activations of the last epoch
            if epoch == nb_epoch - 1 :
                X[:,t] = np.vstack((1,u,x))[:,0]

        #plot some signals to see if IP works
        plot_activity(recorded_res_out, epoch)
        plot_neuron_activity(neuron_out_records, epoch)

    # train the output
    X_T = X.T
    # use ridge regression (linear regression with regularization)
    Wout = np.dot( np.dot(Yt,X_T), np.linalg.inv(np.dot(X,X_T)) + \
        reg*np.eye(1+inSize+resSize) )
    #Wout = np.dot( np.dot(Yt,X_T), linalg.inv( np.dot(X,X_T) + \
    #    reg*eye(1+inSize+resSize) ) )
    # use pseudo inverse
    #Wout = dot( Yt, linalg.pinv(X) )

    #instanciate mean and std to z-score
    mean = np.zeros((outSize,1))
    std = np.ones((outSize,1))
    # run the trained ESN. no need to initialize here,
```

```
# because x is initialized with training data and we continue from there.
Y = np.zeros((outSize,testLen))
u = np.reshape(y_train[trainLen-1],(y_train[trainLen-1].shape[0],1))
for t in range(testLen):
    #compute reservoir outputs
    if ip_mode == 'intrinsic_plasticity_on':
        x = (1-a)*x + a*np.tanh( ip_gain * (np.dot( Win, np.vstack((1,u)))\
        + np.dot( W, x )) + ip_bias )
    else :
        x = (1-a)*x + a*np.tanh(np.dot( Win, np.vstack((1,u)) ) \
        + np.dot( W, x ))
    #compute network outputs
    y = np.dot( Wout, np.vstack((1,u,x)))
    #store network activations
    Y[:,t] = y[:,0]
    u = np.reshape(u_test[t],(u_test[t].shape[0],1))

#z-score network outputS
mean[:,0] = Y.mean(axis=1)
std[:,0] = Y.std(axis=1)
Y = (Y - mean) / std

#instanciate counters to compute verb agreement score
nb_verbs = 0.
nb_verbs_hit = 0.
#instanciate arrays to store the first 6 word predictions
recorded_target_word = []

#compute verb agreement score and plot 6 word predictions
for i in range(testLen):
    #compute the target word
    target_word = compute_word(y_test[i])
    if i<6:
        #collect 6 network activations to plot
        recorded_target_word.append(target_word)
        averaged_recorded_prediction[i,:] += Y[:,i]

    #if target word is a verb, store material to compute agreement score
    if verbs.__contains__(target_word) :
        nb_verbs += 1
        #compute the predicted verb class
        #(singular or plural) given this network output
        verb_class_prediction = compute_class(Y[:,i])

        if singular_verbs.__contains__(target_word) \
           & (verb_class_prediction == 'singular'):
            #that's a hit, the network predicted a singular verb
            nb_verbs_hit += 1
```

```
                if plural_verbs.__contains__(target_word) \
                   & (verb_class_prediction == 'plural'):
                    #that's a hit, the network predicted a plural verb
                    nb_verbs_hit += 1

        #compute MSE for the first errorLen time steps
        errorLen = 500
        mse += sum(sum( np.square( y_test[0:errorLen,:] - Y.T[0:errorLen,:] ))) \
               / errorLen

        #verb agreement rate
        agreement_rate += nb_verbs_hit / nb_verbs

    #average the results over all networks
    print 'MSE = ' + str( mse / nb_network )
    print 'verb agreement rate :' + str(agreement_rate / nb_network)

    #plot the 6 first word prediction during the test set
    plot_predictions(recorded_target_word,
                     averaged_recorded_prediction / nb_network)

    if ip_mode == 'intrinsic plasticity on':
        #plot the evolution of gain and bias to see if they converge
        figure(10).clear()
        plot( record_ip_gain / nb_network, label='gain' )
        plot( record_ip_bias / nb_network, label='bias' )
        legend()
        ylabel('mean value')
        xlabel('number of timesteps')
        title('Evolution of the mean of gain and bias '\
        'relative to intrinsic plasticity')

    show()
```

Références

[1] H. Jaeger(2002) *A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach*

[2] David Verstraeten *Reservoir Computing : computation with dynamical systems.*

[3] Mantas Lukoševičius (2012) *A Practical Guide to Applying Echo State Networks.* G. Montavon, G. B. Orr, and K.-R. M uller, editors, Springer

[4] Jochen J. Steil (2007) *Online reservoir adaptation by intrinsic plasticity for backpropagation–decorrelation and echo state learning.* Neural Networks 20 (2007) 353–364 Special Issue

[5] Jochen J. Steil (2008) *Improving reservoirs using intrinsic plasticity.* Neurocomputing 71 (2008) 1159–1171 Special Issue

[6] Stemmler M. & Koch C. (1999). *How voltage-dependent conductances can adapt to maximise the information encoded by neural firing rate.* Nature Neuroscience, 2, 521–527.

[7] Elman, J. L. (1993). *Learning and development in neural networks : The importance of starting small* Cognition, 48, 71–99.

[8] Matthew H. Tong, Adam D. Bickett, Eric M. Christiansen, Garrison W. Cottrell (2007) *Learning grammatical structure with Echo State Networks.* Neural Networks 20 (2007) 424–432 Special Issue

[9] Baddeley R., Abbott L. F., Booth M. C. A., Sengpiel F., Freeman T., Wakeman E. A. (1997) *Proceedings of the Royal Society of London. Series.*B, 264(1389), 1775–1783. 1997

[10] Arnaud Rachez `https://github.com/zermelozf/esn-lm`